Jessie Plugin and ACSL Specifications

Virgile Prevosto

CEA List

March 31, 2009

Software Analyzers

Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

- Example 1: Searching
- Example 2: Sorting

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq/Isabelle)

Usage

- Proof of functional properties of the program
- Modular verification (function per function)

Limitations

- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care

What is Jessie Useful for?

Usage

- Proof of functional properties of the program
- Modular verification (function per function)

Limitations

- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care

Usage

- Proof of functional properties of the program
- Modular verification (function per function)

Limitations

- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care

What is Jessie Useful for?

Usage

- Proof of functional properties of the program
- Modular verification (function per function)

Limitations

- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care

What is Jessie Useful for?

Usage

- Proof of functional properties of the program
- Modular verification (function per function)

Limitations

- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care

What is Jessie Useful for?

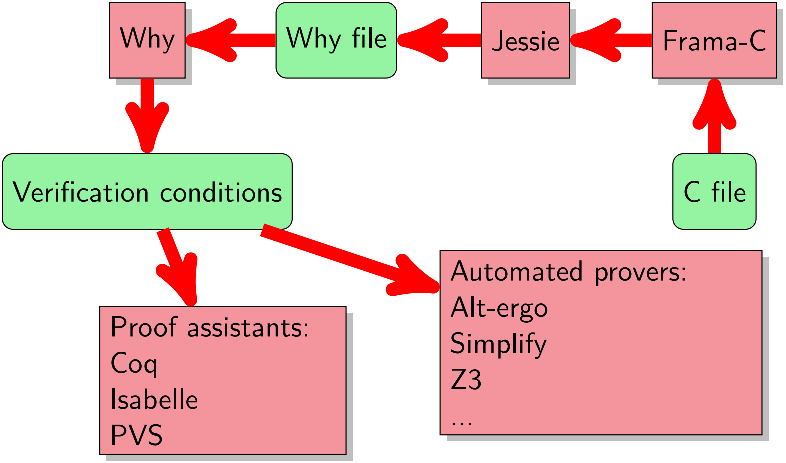
Usage

- Proof of functional properties of the program
- Modular verification (function per function)

Limitations

- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care

From Frama-C to Theorem Provers



A first example

Check safety of a function

- Pointer accesses
- Arithmetic overflow
- Division

```
unsigned int M;

void mean(unsigned int* p, unsigned int* q) {
  M = (*p + *q) / 2;
}
```

Frama-C
Software Analyzers

Behavior of a function

- Functional specification
- Pre-conditions (requires)
- Post-conditions (ensures)

Example

```
unsigned int M;
/*@
  requires \valid(p) ∧ \valid(q);
  ensures M ≡ (*p + *q) / 2;
*/
void mean(unsigned int* p, unsigned int* q) {
  if (*p ≥ *q) { M = (*p − *q) / 2 + *q; }
  else { M = (*q − *p) / 2 + *p; }
}
```

Behavior of a function

- Functional specification
- Pre-conditions (requires)
- Post-conditions (ensures)

Example

```
unsigned int M;
/*@
  requires \valid(p) ∧ \valid(q);
  ensures M ≡ (*p + *q) / 2;
*/
void mean(unsigned int* p, unsigned int* q) {
  if (*p ≥ *q) { M = (*p − *q) / 2 + *q; }
  else { M = (*q − *p) / 2 + *p; }
}
```

Behavior of a function

- Functional specification
- Pre-conditions (requires)
- Post-conditions (ensures)

Example

```
unsigned int M;
/*@
  requires \valid(p) ∧ \valid(q);
  ensures M ≡ (*p + *q) / 2;
*/
void mean(unsigned int* p, unsigned int* q) {
  if (*p ≥ *q) { M = (*p − *q) / 2 + *q; }
  else { M = (*q − *p) / 2 + *p; }
}
```

The specification:

```
/*@
  requires \valid(p) ∧ \valid(q);
  ensures M ≡ (*p + *q) / 2;
  assigns M;
*/
void mean(unsigned int* p, unsigned int* q);
```

- Jessie Usage
- Function Contracts
- **Generating Proof Obligations**
- Advanced Specification
  - Example 1: Searching
  - Example 2: Sorting

- Introduced by Floyd and Hoare (70s)
- Hoare triple: $\{P\}s\{Q\}$, meaning: *If P holds, then Q will hold after the execution of statement s*
- Deduction rules on Hoare triples: *Axiomatic semantic*

Some rule examples

$$\frac{}{\{P\}\texttt{\{\}}\{P\}} \qquad \frac{P \Rightarrow P' \qquad \{P'\}\texttt{s}\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\texttt{s}\{Q\}}$$

$$\frac{\{P\}\texttt{s\_1}\{R\} \qquad \{R\}\texttt{s\_2}\{Q\}}{\{P\}\texttt{s\_1;s\_2}\{Q\}} \qquad \frac{e \text{ evaluates without error}}{\{P[x \leftarrow e]\}\texttt{x=e;}\{P\}}$$

$$\frac{\{P \land \texttt{e}\}\texttt{s\_1}\{Q\} \qquad \{P \land \texttt{!e}\}\texttt{s\_2}\{Q\}}{\{P\}\texttt{if (e) \{ s\_1 \} else \{ s\_2 \}}\{Q\}}$$

$$\frac{\{I \land \texttt{e}\}\texttt{s}\{I\}}{\{I\}\texttt{while (e) \{ s \}}\{I \land \texttt{!e}\}}$$

Weakest pre-condition

- Program seen as a predicate transformer
- Given a function s, a pre-condition Pre and a post-condition $Post$
- We start from $Post$ at the end of the function and go backwards
- At each step, we have a property Q and a statement s, and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- When we reach the beginning of the function with property P, we must prove $Pre \Rightarrow P$.

frama C
Software Analyzers

Weakest pre-condition

- Program seen as a predicate transformer
- Given a function s, a pre-condition *Pre* and a post-condition *Post*
- We start from *Post* at the end of the function and go backwards
- At each step, we have a property Q and a statement s, and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- When we reach the beginning of the function with property P, we must prove $Pre \Rightarrow P$.

Weakest pre-condition

- Program seen as a predicate transformer
- Given a function s, a pre-condition *Pre* and a post-condition *Post*
- We start from *Post* at the end of the function and go backwards
- At each step, we have a property Q and a statement s, and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- When we reach the beginning of the function with property P, we must prove $Pre \Rightarrow P$.

Weakest pre-condition

- Program seen as a predicate transformer
- Given a function s, a pre-condition *Pre* and a post-condition *Post*
- We start from *Post* at the end of the function and go backwards
- At each step, we have a property Q and a statement s, and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- When we reach the beginning of the function with property P, we must prove $Pre \Rightarrow P$.

Weakest pre-condition

- Program seen as a predicate transformer
- Given a function s, a pre-condition *Pre* and a post-condition *Post*
- We start from *Post* at the end of the function and go backwards
- At each step, we have a property Q and a statement s, and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- When we reach the beginning of the function with property P, we must prove $Pre \Rightarrow P$.

- Assignment

$$WP(\texttt{x=e}, Q) = Q[x \leftarrow e]$$

- Sequence

$$WP(\texttt{s\_1;s\_2}, Q) = WP(\texttt{s\_1}, WP(s_2; Q))$$

- Conditional

$$WP(\texttt{if (e) \{ s\_1 \} else \{ s\_2 \}}, Q) = \\ \texttt{e} \Rightarrow WP(\texttt{s\_1}, Q) \wedge \texttt{!e} \Rightarrow WP(\texttt{s\_2}, Q)$$

- While

$$WP(\texttt{while (e) \{ s \}}, Q) = \\ I \wedge \forall \omega . I \Rightarrow (\texttt{e} \Rightarrow WP(\texttt{s}, I) \wedge \texttt{!e} \Rightarrow Q)$$

Memory Model

Issue

How can we represent memory operations (`*x`, `a[i]=42`,. . .) in the logic

- If too low-level (a big array of bytes), proof obligations are intractable.
- If too abstract, some C constructions can not be represented (arbitrary pointer casts, aliasing)
- Standard solution (Burstal-Bornat): replace struct's components by a function

Aliasing

Issue

The same memory location can be accessed through different means:

```
int y;
int* yptr = &y;
*yptr = 3;
/*@ assert y ≡ 3; */
```

- Again, supposing that any two pointers can be aliases would lead to intractable proof obligations.
- Memory is separated in disjoint regions
- Some hypotheses are done (as additional pre-conditions)

Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

- Example 1: Searching
- Example 2: Sorting

A concrete example

Informal spec

- Input: a sorted array and its length, an element to search.
- Output: index of the element or −1 if not found

Implementation

```c
int find_array(int* arr, int length, int query) {
  int low = 0;
  int high = length - 1;
  while (low <= high) {
    int mean = low + (high -low) / 2;
    if (arr[mean] == query) return mean;
    if (arr[mean] < query) low = mean + 1;
    else high = mean - 1;
  }
  return -1;
}
```

An example

Informal specification

- Input: an array and its length
- Output: the array is sorted in ascending order

```
int index_min(int* a, int low, int high);

void swap(int* arr, int i, int j);

void min_sort(int* arr, int length) {
  for(int i = 0; i < length; i++) {
    int min = index_min(arr,i,length);
    swap(arr,i,min);
  }
}
```